Report Number: WUCS-96-24

1996-01-01

End-User Construction and Configuration of Distributed Multimedia Applications

Terrance Paul McCartney

Distributed multimedia applications supported by a global electronic infrastructure have tremendous potential for providing users with customized communication and computation environments. Since communication and computation requirements vary by context and change dynamically, it is unlikely that off-the-shelf applications will anticipate the needs of all users. Therefore, empowering end-users to create their own customized applications for both communication and computation is an important challenge. This dissertation presents several mechanisms that enable end-users to create and configure distributed multimedia applications, including end-users construction direct manipulation graphical users interface (GUIs) and application management of distributed multimedia applications over the Internet.
... Read complete abstract on page 2.

End-User Construction and Configuration of Distributed Multimedia Applications

Terrance Paul McCartney

Complete Abstract:

Distributed multimedia applications supported by a global electronic infrastructure have tremendous potential for providing users with customized communication and computation environments. Since communication and computation requirements vary by context and change dynamically, it is unlikely that off-the-shelf applications will anticipate the needs of all users. Therefore, empowering end-users to create their own customized applications for both communication and computation is an important challenge. This dissertation presents several mechanisms that enable end-users to create and configure distributed multimedia applications, including end-users construction direct manipulation graphical users interface (GUIs) and application management of distributed multimedia applications over the Internet.

End-User Construction and Configuration of Distributed Multimedia Applications

Terrance Paul McCartney

WUCS-96-24

December 1996

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
St. Louis MO 63130

A dissertation presented to the Sever Institute of Washington University in partial fulfillment of the requirements of the degree of Doctor of Science, December, 1996, Saint Louis, Missouri.

WASHINGTON UNIVERSITY
SEVER INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

END-USER CONSTRUCTION AND CONFIGURATION OF DISTRIBUTED MULTIMEDIA APPLICATIONS

by

Terrance Paul McCartney

Prepared under the direction of Professor Kenneth J. Goldman

A dissertation presented to the Sever Institute of
Washington University in partial fulfillment
of the requirements of the degree of

DOCTOR OF SCIENCE

December, 1996

Saint Louis, Missouri

WASHINGTON UNIVERSITY
SEVER INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

ABSTRACT

END-USER CONSTRUCTION AND CONFIGURATION OF DISTRIBUTED MULTIMEDIA APPLICATIONS

by Terrance Paul McCartney

ADVISOR: Professor Kenneth J. Goldman

December 1996

St. Louis, Missouri

Distributed multimedia applications supported by a global electronic infrastructure have tremendous potential for providing users with customized communication and computation environments. Since communication and computation requirements vary by context and change dynamically, it is unlikely that off-the-shelf applications will anticipate the needs of all users. Therefore, empowering end-users to create their own customized applications for both communication and computation is an important challenge. This dissertation presents several mechanisms that enable end-users to create and configure distributed multimedia applications, including end-user construction direct manipulation graphical user interfaces (GUIs) and application management of distributed multimedia applications over the Internet.

This dissertation is done in the context of *The Programmers' Playground,* a software library and run-time system for creating distributed multimedia applications. The focus of this dissertation is the human-computer interaction (HCI) aspects of The Programmers' Playground, allowing end-users to construct and use distributed multimedia applications without the need for programming, compiling, writing scripts, or interacting with the UNIX shell. HCI contributions include a multi-way constraint algorithm, constraint-based editing & visualization, separation of applications from GUIs, support for the construction of multi-user applications, end-user definable widgets, end-user definable aggregate mappings, and World Wide Web-based application management.

To my parents, who always believed in me.

Contents

Figures

Tables

Glossary

Application Management - Coordination of information used for end-user documentation, informed module launching, and configuration of distributed multimedia applications.

Configuration - The communication structure of an application. In this dissertation, the configuration refers to the constraint relationships among user interface graphics and/or the logical connections among distributed application modules.

Constraint - A relationship to be maintained among a set of variables. Usually, this relationship is satisfied by computing the value of an *output variable* as a function of a set *input variables*. In this dissertation, "constraint" refers to dataflow constraints computed using local propagation.

Data Boundary - In Playground, "data boundary" refers to a module's set of published variables (also known as the "presentation"). For a widget, "data boundary" refers its set of exposed attributes.

Distributed Application - An application consisting of multiple concurrent processes, usually running on separate workstations, that communicate over a network. In this dissertation, an application consists of a collection of independent modules and a configuration of logical connections among the modules' published variables.

End-user - The user of an application. In this dissertation, end-users are people who are proficient in WYSIWYG applications, such as word processors and graphics editors, but are not necessarily experienced in textual programming.

EUPHORIA - End-User Production of graphical interfaces fOR Interactive distributed Applications

I/O Abstraction - A connection-oriented model of interprocess communication in which independent modules interact with an abstract environment.

Logical Connection - The communication specification between two published variables. Connections can specify either bidirectional or unidirectional communication.

Module - A component of a distributed application. In this dissertation, a module is a process that has a set of published variables.

Multimedia - Integration of different media types such as text, interactive graphics, images, audio, and video within an application. In this dissertation, multimedia also includes communication and coordination using the different media types.

Multi-way Constraint - A constraint whose computation direction may dynamically change based on the addition or deletion of other constraints to a constraint graph. That is, the choice of computation function, input variables, and output variable change over time.

One-way Constraint - A constraint whose computation is static, always computing the same output variable in terms of the same set of input variables.

Published Variable - A data structure, such as a real number or an array, that is exposed from a module to the external environment. Whenever a published data structure is updated, communication to other published variables occurs implicitly according to the outgoing logical connections.

User Interface Management System (UIMS) - A tool for developing graphical user interfaces, typically through the use of a visual language in a graphical environment. Different from an *interface builder*, which is limited to "dialog box" style user interfaces, a UIMS provides more comprehensive support for application-specific graphics and behavior.

Acknowledgments

Ken Goldman has been an excellent advisor, providing encouragement, advice, and innovative ideas that enhanced the overall quality of this dissertation and the associated software systems. Ken is always willing and eager to talk about research problems and other issues whenever I wander into his office. His presence has a way of attracting bugs like a bright flame on a dark summer night. I also thank the rest of my doctoral committee, Michael Kahn, Dan Kimura, Eileen Kraemer, and Brad Myers, for their insightful comments and suggestions.

I thank Bala Swaminathan who developed the Playground Protocol, Veneer, and Connection Manager, that my software uses. I also thank Ram Sethuraman who undertook the thankless job of maintaining the Playground library, fixing the bugs that came out to bite me. David Saff helped a great deal in the implementation of EUPHORIA, implementing constraint visualization, calculators, as well as miscellaneous parts of the graphics toolkit. I also thank Trevor Harmon for his work in porting EUPHORIA to Windows NT.

Several people, as well as the students of Washington University's CS333 Spring 1995 and CS333 Spring 1996 courses, have used EUPHORIA to develop distributed multimedia applications. Some of these applications are used as examples in this dissertation. I thank Mitch Reierson for developing the Hypervideo Browser, Scott MacDonald for developing the Remote Control Camera Application, and David Saff for developing the Image Morphing Application.

I thank Karl Schmidt, Jyoti Parwatikar, and Eileen Kraemer who volunteered to be "pre-test" subjects for my usability study. They patiently followed the instructions of the study, helping me to identify ambiguous wording in the instructions. Their help greatly contributed to the relatively problem-free testing of the real subjects.

Berkley Shands and Bill Shapiro implemented the Application Management System from my design, fighting the evils of UNIX system calls and Netscape Applet security. My graduation would have been significantly delayed if I had to do this implementation myself.

I thank my parents, Terry and June, for all of their love and support over the years of my education. They were always encouraging, even when I doubted myself.

This research was supported in part by the Federal Graduate Workstudy program, National Science Foundation grants CCR-91-10029, CCR-94-12711, and ARPA contract DABT63-95-C-0083.

Chapter 1

Introduction

Distributed multimedia applications supported by a global electronic infrastructure have tremendous potential for providing users with customized communication and computation environments. Applications include remote collaboration, information and resource sharing, and access to broadcast media. Future users of the infrastructure will vary greatly in technical skills, ranging from novice users to sophisticated expert users and programmers. Since communication and computation requirements vary by context and change dynamically, it is unlikely that off-the-shelf applications will anticipate the needs of all users. Therefore, empowering end-users to create their own customized applications for both communication and computation is an important challenge. Support for end-user construction of distributed applications means not only that users can combine software components, but also that they can construct graphical user interfaces for interaction with these custom applications. A complete solution to the end-user construction problem requires both user interface management for end-user construction of direct manipulation graphical user interfaces and application management for end-user configuration of distributed applications.

1.1 Problem Statement

For the purposes of this dissertation, an *end-user* is defined as a person who can operate a computer and use WYSIWYG applications such as a graphics editor, but cannot necessarily write textual computer programs. Currently, it is very difficult for end-users to construct complete distributed multimedia applications due to the technical challenges associated with each development stage. The task of constructing a distributed multimedia application involves four

distinct stages: (1) creation of the application portion, (2) construction of one or more direct manipulation graphical user interfaces, (3) establishing interaction among the application and its GUIs, and (4) enabling others to use the application. The next four subsections discuss the specific problems that confront researchers in supporting end-user activity in each of these four areas.

1.1.1 Application Portion

The "application portion" of a distributed multimedia application refers to the application's underlying computation independent of its graphical representation. Creating the application portion generally requires sophisticated technical skills. First, a solid knowledge of concurrency principles is needed to properly reason about a distributed application during its design. Most end-users are unlikely to comprehend these principles without a great deal of computer science education. Second, specification of an application's computation generally requires some form of textual programming. Although it may be possible to apply visual programming techniques in creating the application portion, we believe that this approach is not feasible for large-scale, general purpose computation. Large visual programs are generally difficult to follow because of limited screen space. Also, pure visual programming typically limits the programmer to a small set of predefined primitives, restricting the types of applications that may be constructed. End-users must be given the ability to assemble the application portion for a wide range of application domains without being exposed to the complexities of textual programming or concurrency issues.

1.1.2 GUI Construction

"Direct manipulation graphical user interface" (GUI) refers to a user interface with application-specific graphics, animation, and state-based interaction (e.g., Interactive Orbital Simulation, Section 1.6.1). GUI development is a difficult task, requiring a great amount of time at each stage of the software lifecycle. Brad Myers stated this difficulty well in his article titled "User Interface Software Tools" [47]:

> User interface software is often large, complex and difficult to implement, debug, and modify. One study found that an average of 48% of the code of applications is devoted to the user interface, and that about 50% of the implementation time is devoted to implementing the user interface portion [49]. As interfaces become easier to use, they become harder to

> create [45]. Today, direct manipulation interfaces are almost universal: one 1993 study found that 97% of all software development on UNIX involved a GUI [82]... Tomorrow's user interfaces will provide speech and gesture recognition, intelligent agents and integrated multimedia, and will probably be even more difficult to create. Furthermore, because user interface design is so difficult, the only reliable way to get good interfaces is to interactively re-design (and therefore re-implement) the interfaces after user-testing, which makes the implementation task even harder.

Previous research and software systems for aiding user interface construction fall into two general categories: textual specification systems (e.g., graphics packages, toolkits, etc.), and graphical specification systems. Textual specification systems such as MacApp [80] and InterViews [37] are inappropriate for end-users since extensive textual programming expertise is required. For example, to master a graphics toolkit one must be familiar with dozens of function calls and data structures. Graphical specification systems are an excellent start toward providing the end-user with the ability to construct a GUI since the appearance may be specified interactively without textual programming.

In constructing a GUI, one major source of difficulty is maintaining consistency among user interface state values. Usually, state values can be changed in multiple ways (e.g., by direct manipulation or by the application portion), can have multiple graphical representations, and can have side-effect actions associated with modification. Also, there may be dependencies among state values, requiring potentially many updates and side-effects to be performed as the result of changing a single state value in a particular way. For highly interactive user interfaces, programmers spend much of their time hard-coding the source code for special cases of state modification. This tends to be a highly error prone process since it can be difficult to enumerate all possible modification scenarios. One approach to solving this problem is to use a state transition tool such as the VAPS system [78]. However, transition networks have not had much success since they do not scale well and are unintuitive to end-users. In dynamically changing GUIs, all of the possible state components may not be known in advance, ruling out the possibilities of hand-crafting the source code and state transition networks.

1.1.3 Application-GUI Integration

Once the application portion and its GUIs are constructed, integration is necessary. Enabling end-users to specify the interaction between a user interface and its underlying application has been a long standing goal of user interface research. It is very difficult to give this power to the end-user since, for non-trivial distributed applications, the application portion consists almost entirely of textual source code that the end-user is unlikely to understand. Indeed, most user interface development systems only allow the programmer to specify the interaction between the application and its GUIs, typically by filling in code stubs with function calls directly to application code. This approach is also undesirable since it tends to tightly couple the application portion to its graphical representation, making it less portable and difficult to maintain [73]. Ideally, end-users should be given the ability to integrate the application portion, specified textually by programmers, with their own customized direct manipulation GUIs, created through the use of a graphical specification system.

1.1.4 Application Management

Distributed multimedia applications consist of multiple components (e.g., application portion, GUIs) with varying configurations and performance requirements. To use a completed application, this information must be used to ensure correct and optimal performance. For example, there may be the requirement that the application portion must operate on a particular file system in order to access a locally stored database. Exposing this information directly to the end-user is undesirable since it forces the end-user to be knowledgable about the application's internal details and the current performance characteristics of the resources that may be used. To enable end-users to use a distributed application, it is necessary to automate the process of launching and configuration through the use of an "application management" mechanism.

1.2 Hypothesis

It is possible to provide software support that enables end-users to:

1. construct direct manipulation graphical user interfaces for visualization and direct manipulation of distributed application state,
2. specify both the intraprocess and interprocess communication of distributed multimedia applications, and
3. use completed distributed applications.

This work attacks the problem in three subproblem areas: (1) user interface construction, (2) constraints, and (3) application management, as described below.

1.3 Approach

The following three subsections outline the approach taken in solving the three subproblem areas.

1.3.1 User Interface Construction

As discussed in Section 1.1, constructing a distributed multimedia application involves three stages: creating the application portion, specifying direct manipulation graphical user interfaces, and integrating the GUIs with their underlying application portion.

Application Portion

Traditionally, distributed application development has been an extremely difficult and complicated task. Only programmers having a solid understanding of concurrency and interprocess communication were able to construct distributed applications. Applications developed in this way provided users with little flexibility or means for customization. *The Programmers' Playground* [19], [21] is a distributed programming environment for the construction of distributed multimedia applications. Playground provides a software library and runtime system, allowing distributed applications to be constructed from a number of independent components called *modules*. Through the use of the Playground library and runtime system, the details of interprocess communication are completely hidden from the implementors of the distributed

application's modules. In this way, programmers can develop the components of distributed applications without knowing about operating system primitives such as sockets and semaphores.

We utilize The Programmers' Playground for the development of the application portion of distributed multimedia applications. General-purpose, off-the-shelf modules can be developed relatively easily by programmers. Playground modules are designed to permit customization of distributed applications through dynamic runtime configuration of modules. Configuration of distributed applications involves end-users taking a number of these independent off-the-shelf modules and establishing the communication structure dynamically at runtime through a visual language that we have developed (Section 1.5.1). Playground was chosen over other distributed application systems such as CORBA [77] because its connection-oriented semantics is more intuitive to end-users and maps efficiently onto connection-oriented networks such as ATM.

GUI Specification

Ideally, end-users should be able to specify and change the application-specific appearance and behavior of a GUI. Graphical specification systems allow the end-user to draw the appearance of a GUI, accelerating the development process. Many such previous environments, such as DialogEditor [8] and Prototyper [68], only support limited classes of GUIs (e.g., dialog boxes) and do not provide support for construction of application-specific data visualization and manipulation. While systems such as Peridot [46] and Lapidary [50] allow basic behavior and widgets to be defined through demonstration, these systems do not give the end-user the ability to integrate the application portion with the user interface portion.

To explore user interface construction techniques, we implemented a user interface management system called *EUPHORIA* that allows end-users to specify direct manipulation graphical user interfaces using a graphics editor. This approach builds upon the end-user's prior experience with other graphics editors, resulting in much less of a learning curve. In addition to simply drawing the appearance of the GUI, the end-user can also establish relationships among user interface components through direct manipulation. These relationships include constraints, encapsulation, and aggregate-based visualization and manipulation. Relationships are specified explicitly using a connection-oriented visual language, rather than by demonstration.

Application-GUI Integration

A distributed application is integrated with its GUI through the use of The Programmers' Playground published data structures (see Section 1.5.1). In this way, the application portion can be written independent from its graphical representation (e.g., no explicit calls are needed to redraw the window when the state has changed). EUPHORIA is implemented as a Playground module, allowing important state components of a GUI to be published for communication with external Playground modules. These state components can be configured to the application portion by end-users through the use of a visual configuration mechanism.

1.3.2 Constraints

Constraints are widely recognized as a useful tool for maintaining relationships in user interfaces [6], [18], [26], [39], [62], [63], [65], [76]. State components, dependencies, and side-effects can be defined simply through the use of constraints, leaving the task of maintaining these relationships to a general purpose constraint solver. Constraints are particularly appealing for end-user construction of GUIs since they have an intuitive "connection oriented" semantics that is easy to understand.

Multi-way constraint solvers that support constraint hierarchies have proven to be useful in the development of user interfaces, enabling dynamically changing relationships to be defined using preference levels [65], [75]. However, previous such constraint solvers lacked the ability to effectively handle cyclic constraint relationships. Constraint cycles commonly arise when the end-user is given the power to establish constraint relationships. Failure to resolve constraint cycles in a reasonable way can result in unreliable and unpredictable behavior. Previous constraint solvers either did not allow cycles of constraints to be formed, or maintained cycles of constraints and used a separate system to satisfy each cycle's constraints. Solvers that maintain cycles of constraints have a few problems. First, the problem of maintaining cyclic hierarchical constraint graphs is NP-complete [39], so there is no efficient optimal solution. Second, it is possible to form cycles of constraint relationships that do not have solutions. Third, these solvers typically limit each constraint's computation type (e.g., only linear equations), rather than allowing general-purpose computation. Acyclic solvers generally break cycles when encountered by arbitrarily

leaving a constraint on the cycle unenforced. This approach reduces the number of constraints in the system that may be satisfied, and does not respect the relative preference levels specified with the constraint hierarchy. For example, a more important constraint may be left unenforced in order to satisfy a constraint of only minor importance.

To automate intraprocess communication and to maintain consistency among user interface state components, we have developed *UltraBlue*, a multi-way constraint solver algorithm. UltraBlue supports constraint hierarchies and avoids cyclic constraint relationships with respect to the constraints' relative preference levels. This approach provides end-users with fast performance needed for interactive applications and also makes the behavior of a GUI more reliable and predictable.

1.3.3 Application Management

The application management mechanism must be able to launch and configure the modules of a distributed application on a variety of workstations and should be available to a large number of end-users. For these reasons, we have designed a distributed resource allocation system and a World Wide Web [2] interface, allowing Internet users to find and launch distributed multimedia applications using an existing web browser (e.g., Netscape Navigator [51]). The resource allocation system was developed as a Playground application and the World Wide Web interface was developed as a Java applet [1].

Other systems such as Prospero [52] have been developed for resource allocation in a distributed environment. However, these systems assume that the configuration among the components is handled by the application. Our system includes a mechanism for configuring new applications and joining existing applications, freeing the application designer from the burden of hard-coding a configuration mechanism into the application.

1.4 Contributions

Contributions of this dissertation include the following:

1. Interactive techniques enabling end-users to create and modify constraint relationships among graphics components including direct constraint relationships (equality, conversion, anchor), indirect constraint relationships (imaginary objects, formula constraints), and constraint visualization.

2. Interactive techniques enabling end-users to create compound graphics objects (i.e., widgets) with user defined coordinate systems, data interfaces, and alternative representations.

3. An end-user mechanism for the visualization and manipulation of aggregate data, including mechanisms similar to a relational database's project, select, join, and cross product operations.

4. An incremental, hierarchical, multi-way constraint algorithm that supports simple inequality relationships and resolves cycles of constraints with respect to the relative preference level of each constraint.

5. An application management system that enables end-users to launch and configure a distributed multimedia application through interaction with a World Wide Web based interface.

6. An empirical study of end-users that demonstrates the usability of our application construction techniques.

1.5 Background

This section presents a brief background of previous work. Later chapters assume that the reader is familiar with this information and terminology.

1.5.1 The Programmers' Playground

The Programmers' Playground [19], [20], [21] is a software library and run-time system that supports a new programming model for distributed applications. The model, called *I/O abstraction*, provides a separation of computation from communication that is well-suited for end-user construction of customized distributed applications from computational building blocks. Playground users do not need to write any source code to establish communication between the

modules of a distributed application, nor do they need to understand the details of how communication occurs. This section provides an overview of Playground and the I/O abstraction model on which it is based. Details on Playground may be found elsewhere [20], [21].

In the I/O abstraction model, each *module* of a distributed application has a *data boundary*[1] containing published variables that may be externally observed and/or manipulated. Modules are written in a standard programming language (e.g., C++) using the Playground library (see Appendix D for an example). This library provides a set of publishable data types, including base types (e.g., integer, real, string), tuples, and aggregates (e.g., arrays, sets). Programmers may arbitrarily nest these types to form new publishable data types, and new publishable aggregates may be defined as well.

A distributed application consists of a collection of independent modules and a configuration of *logical connections* among the published variables in the module data boundaries. Whenever a module updates one of its own published data items, the new value is implicitly communicated to all connected variables in other modules. The details of how the communication is handled are hidden from the implementor and users of the module. This simplifies module construction and gives the run-time system flexibility in optimizing communication. The configuration of connections is determined dynamically at run-time, rather than statically at compile time. This gives end-users the flexibility to add new components or relationships to their applications dynamically.

I/O abstraction communication is declarative, rather than imperative. That is, one declares high-level logical connections among the data items of individual modules, as opposed to directing communication within the control flow of the module. Output is implicit, a by-product of computation. Input is observed passively, or handled by reactive control within a module. This declarative approach simplifies application programming by cleanly separating computation from communication. Software modules written using I/O abstraction do not make explicit requests to

1. In other papers describing the I/O abstraction concept, the data interface of an I/O abstraction module has been called the "presentation." Since this dissertation deals with user interfaces, we use the term "data boundary" in order to avoid confusion.

establish or effect communication, but instead are concerned only with the details of the local computation. Exposing the configuration also allows the run-time system to handle communication more effectively.

Playground modules have a visual representation that was designed as part of a visual language for interprocess and intraprocess communication [42]. A Playground module is represented as a box with a data "plug" for each variable in the module's data boundary. The color of each variable represents its type. Logical connections are represented as arrows between pairs of variables in module data boundaries. The metaphor is that of wiring together the components of a stereo system, where the color of each cord denotes the type of information that it carries.

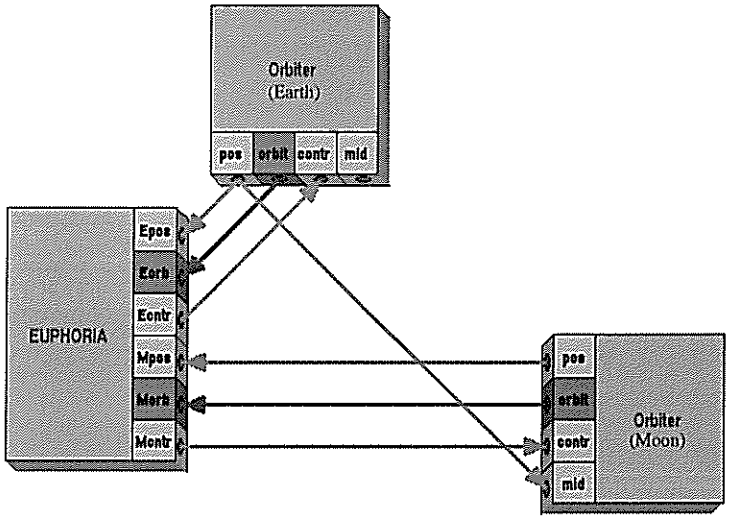


Figure 1-1: Orbital simulation modules.*

For example, Figure 1-1[2] shows the visual representation of the Interactive Orbital Simulation (Section 1.6.1). This application consists of three communicating modules: EUPHORIA and two instantiations of an Orbiter module. The Orbiter modules simulate the position and orbits of the Earth and Moon, respectively, and respond to changes to the graphical controls. Each module has a data boundary of published variables. For example, each Orbiter module has four published variables representing the position, orbit, controls, and orbit mid-point. End-users define logical

2. Color versions of selected figures are included in Appendix A. These figures are denoted by an asterisk by their figure title.

connections among the modules graphically by dragging connection arrows between variables. For example, the position of the Earth's Orbiter module is connected to both EUPHORIA and to the Moon's Orbiter module.

1.5.2 Constraints

A *constraint* is a relationship to be maintained among a set of variables. For example, "A+B=C" represents the relationship that C is the sum of A and B. Through constraints, relationships among user interface components and their applications can be defined declaratively, leaving the task of maintaining the relationships to a constraint solver, an algorithm that determines a plan for computing constrained values in a way that is consistent with the specified constraints. Constraints simplify the programmer's task in creating user interfaces and empower end-users to define sophisticated relationships without the need for programming.

A constraint represents a computation on a set of *input variables* (variables used by the constraint's computation) to an *output variable* (variable computed by the constraint's computation). Each variable may have multiple associated constraints, forming a directed *constraint graph.*

Multi-way Constraints

Multi-way dataflow constraint systems allow constraints to have a dynamically changing computation flow. That is, a constraint's input variables, output variables, and its computation can vary based on the addition or deletion of constraints to the constraint graph. A set of constraints is *conflict-free* if each variable is an output variable of at most one constraint. A multi-way constraint solver redirects constraints (i.e., changing input/output direction) in order to achieve a conflict-free constraint graph. Multi-way constraints allow dynamically changing relationships to be declaratively defined, giving the constraint solver the task of determining the constraint graph's computation flow. This approach is particularly useful in interactive user interfaces, where the flow of data can change based on interaction and direct manipulation (see Section 2.8).

In all constraint figures in this dissertation, each variable is represented as a circle and each constraint is represented as a square with attached undirected edges from input variables and a

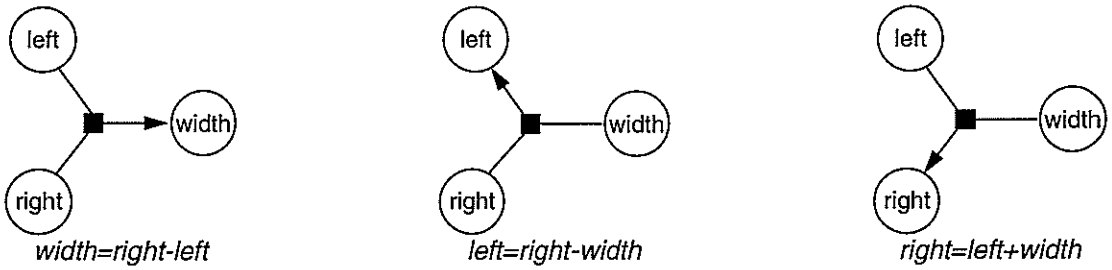


Figure 1-2: Possible computation directions of a multi-way width constraint.

single directed edge to the output variable. Figure 1-2 shows a multi-way constraint representing the relationship "width=right-left." This constraint can be directed to compute any one of its associated variables using an algebraic rearrangement of the constraint's computation. That is, left can be computed with "left=right-width," and right can be computed using "right=left+width." Specifying this relationship as a multi-way constraint makes it possible to change its computation flow in the constraint graph dynamically, rather than hard-coding only one computation direction.

Constraint Strength

While multi-way constraints simplify the task of specifying constraint relationships, their solutions can be unpredictable at times due to nondeterminism inherent in the specification. There may be many possible ways to satisfy a series of multi-way constraints. Constraint hierarchies [7], [15], [65] allow each constraint to be specified using a preference level, or *strength*, representing its relative importance. This strength information is used to determine how to enforce constraints in the event of conflicts, favoring stronger constraints over weaker ones. For the purposes of this dissertation, the following strengths (from weakest to strongest) will be used for illustration purposes: weak, medium, strong, and required. In practice, only eight to twelve strength levels are typically used.

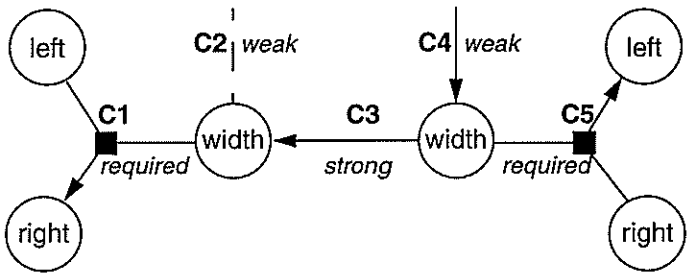


Figure 1-3: Constraint graph with strengths.

For example, Figure 1-3 shows the constraint graph constraining the width of two rectangles to be equal (see Section 2.8.1 for more information). Constraints C1 and C5 represent the relationship "width=right-left," and are specified with a strength of *required*. Constraints C2 and C4 are "stay constraints," serving to keep the widths constant in the absence of other constraints, and are specified with a strength of *weak*. C3 represents equality between the two width variables, and is specified with a strength of *strong*. C1 and C5 are the most important constraints, since they represent geometric truths. C3 is not as important, and can be left unenforced if there is a conflict with a required constraint. C2 and C4 are the least important, and are enforced only if there are no conflicting constraints. Notice that C2 is unenforced (represented as a dashed line) since it conflicts with C3 (recall, a variable can be the output of at most one constraint). Also notice that there is still non-determinism in the choices of constraints' outputs; C1 arbitrarily chooses right as its output while C5 chooses left.

1.6 Examples

Throughout this dissertation, examples are given to motivate key concepts and to illustrate the diverse classes of applications that can be created using our user interface construction techniques. Some of these are "toy applications" presented merely to help explain particular concepts. Section 1.6.1 and Section 1.6.2 presents two example applications, serving as an example-based overview of the user interface construction techniques described in later chapters.

1.6.1 Interactive Orbital Simulation

The Interactive Orbital Simulation (Figure 1-4) is a distributed application that animates the orbits of the Earth and Moon in our solar system. Graphical controls in the user interface allow the end-user to change the simulation properties through direct manipulation. The path of the Earth or Moon can be changed by dragging any of three orbit representative points[3]. Properties such as the speed and rotation direction may also be specified interactively.

---

3. These three points specify the shape of an ellipse. Recall that the sum of the distances from the two foci of an ellipse to any point on the ellipse is constant.

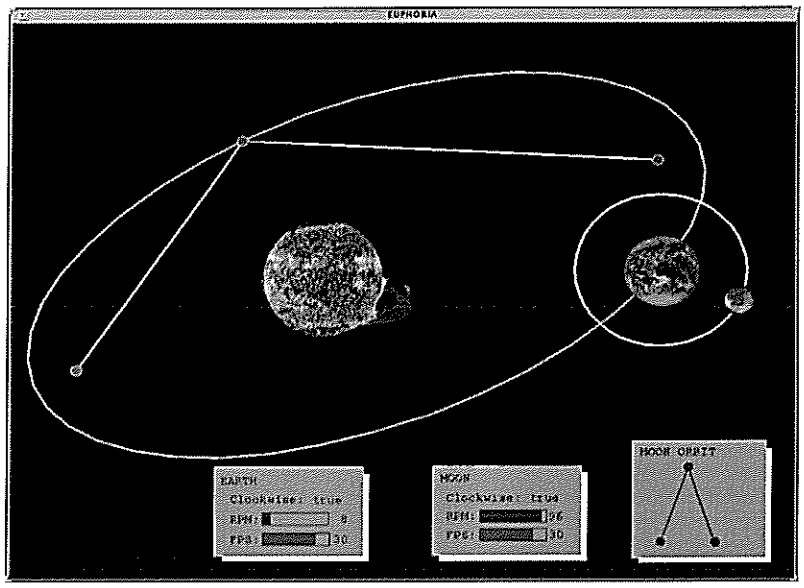


Figure 1-4: Interactive orbital simulation in EUPHORIA.*

This application's graphical interface and the graphical interfaces in other examples in this dissertation were constructed through the use of the EUPHORIA user interface management system. No programming is required to create these GUIs. Instead, end-users simply draw the appearance of the GUIs, establish constraints among the components, and publish relevant state information to the external environment.

This example illustrates the following application construction techniques described within this dissertation. The basic GUI components are drawn using graphics primitives, described in Section 2.4, as well as compound graphics components called *widgets* (e.g., the speed control slider), described in Section 2.7. Constraint relationships among these components are defined graphically by the end-user (e.g., connecting the lines of the orbital controls), as described in Section 2.5. Internally, these constraints are maintained through the use of the UltraBlue constraint algorithm described in Chapter 4.

This application is constructed from three communicating modules, described in Section 1.5.1. Relevant state information (e.g., the position of the Earth) is published to the external environment as described in Chapter 3 and is configured to external application modules through the use of a

visual configuration language as described in Section 1.5.1. This example may also be launched and configured using the Application Management mechanism described in Chapter 5.

1.6.2 Hypervideo Browser

The Hypervideo Browser is a distributed application for viewing related video segments. The semantics of the video segments are displayed using an undirected graph. The vertices of the graph depict places of interest and are shown using representative frames from the video. The edges of the graph form paths connecting vertices of related video segments, enabling quick navigation among places of interest. A token is used to show the current viewing location in the graph; end-users can jump to any video segment by moving the token.

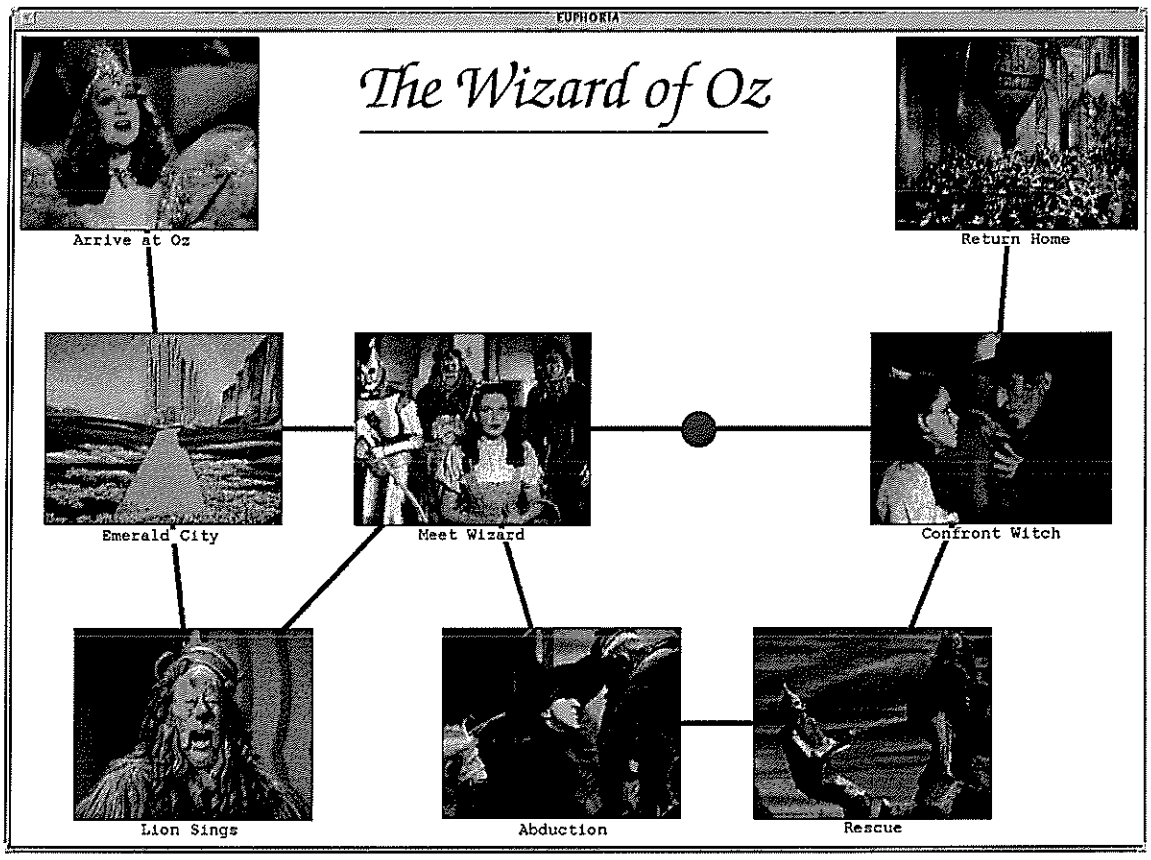


Figure 1-5: Hypervideo Browser in EUPHORIA.*

For example, Figure 1-5 shows a partial graph for the movie "The Wizard of Oz" [81]. One possible path of interest represents the key plot points, and consists of the following vertices:

"Arrive at Oz," "Emerald City," "Meet Wizard," "Confront Witch," and "Return Home." Other less important plot points such as "Abduction" and "Rescue" also exist as a separate path that branches off from the key plot path. Another example path might consist of only the video segments that have music and dancing; users who want to see only these parts of the movie may traverse this path.

Aggregate mappings enable end-users to define interactive visualizations of aggregate elements (see Section 3.5). This example illustrates a few different ways of using aggregate mappings[4]. Since there can be potentially many vertices in the graph, the vertices are stored using an aggregate. An aggregate mapping is used to automate the visualization of each aggregate element, namely the vertices of the graph. The edges of the graph are created with a joined aggregate (see Section 3.5.2), using information from both the vertex aggregate and an edges aggregate. A filter can also be applied on each aggregate mapping (Section 3.5.1) to view only certain parts of the entire hypervideo graph.

1.7 Overview

The remaining chapters of this dissertation are arranged as follows. Chapter 2 discusses graphical user interface construction techniques that empower end-users to construct direct manipulation graphical user interfaces without the need for textual programming. Described in this chapter are graphics primitives (e.g., rectangles, text, images), end-user defined direct constraint relationships (equality, conversion, and anchor constraints), end-user defined indirect constraint relationships (formula constraints and imaginary objects), and widgets. Chapter 3 presents techniques for integrating a user interface management system with a distributed application. The focus of this chapter is communication, visualization, and manipulation of distributed data structures, including base types (e.g., integers, strings), end-user defined tuples, and end-user defined aggregate mappings (e.g., arrays of tuples). Chapter 2 and Chapter 3 each contain a brief description of the constraint architecture used to implement the chapter's user interface construction techniques. Chapter 4 describes UltraBlue, a multi-way constraint algorithm that avoids cycles of constraints

4. As of this writing, the application modules of the Hypervideo Browser are still under development. However, EUPHORIA supports all of the functionality described in this section.

with respect the constraints' relative preference levels. UltraBlue was used in the implementation of most the techniques described in Chapter 2 and Chapter 3. Chapter 5 discusses Application Management of distributed applications, enabling end-users to launch and configure an entire distributed application without the need for writing scripts or interacting with the UNIX shell. Chapter 6 presents the design and results of a usability study experiment of the user interface software described in Chapter 2 and Chapter 3. Chapter 7 concludes with a summary of contributions and a discussion of future research directions.

Chapter 2

Graphical User Interface Construction

User interface development is a difficult task, requiring a great amount of effort at each stage of the software lifecycle [49]. Implementation of a direct manipulation graphical user interface generally requires a great deal of textual programming expertise and knowledge of platform dependent window systems, graphics packages, and toolkits. While many tools have been developed to aid user interface construction [47], these tools typically only support a limited class of user interfaces (e.g., user interfaces composed of a fixed set of predefined components such as buttons and menus) and not generalized direct manipulation user interfaces. In addition, these tools require programming to integrate the user interface with its application. This requirement restricts user interface developers to a selected few individuals (i.e., programmers) rather than end-users, making it difficult to customize or change an application and/or its user interface.

The EUPHORIA user interface management system is based on a visual language we have developed for describing communication among modules in a distributed system and graphics components of a direct manipulation graphical user interface [42]. This chapter describes user interface construction techniques that allow end-users to create simple and compound application-specific graphics and define relationships among the graphics. Chapter 3 discusses how end-users can integrate these graphical components with an underlying distributed application. Appendix B provides the complete reference manual for EUPHORIA.

2.1 Contributions

1. End-user techniques for forming anchor, equality, and conversion constraints among graphics objects and the external environment.
2. Visualization and end-user editing of constraints.
3. Imaginary objects for forming indirect constraint relationships.
4. An end-user mechanism for establishing formula constraints.
5. End-user techniques for defining widgets with independent coordinate systems and alternative representations.

2.2 Motivating Examples

This section presents two example applications that utilize some of the key user interface construction techniques described within this chapter.

2.2.1 Maple Syrup Factory

Maple syrup is produced by pouring maple sap into a vat and boiling away excess water until a suitable concentration level is reached (see Figure 2-1). A factory producing maple syrup must adjust a number of actuators controlling properties such as the incoming sap flow rate and the burner status (i.e., on or off). The Maple Syrup Factory application automates maple syrup production, controlling the factory actuators in response to sensor values. An interactive GUI is created in EUPHORIA that animates the production state, allowing an operator to monitor conditions and to change actuator values.

End-users create GUIs in EUPHORIA through the use of a graphics editor consisting of a Drawing Palette, Data Boundary, and Drawing Area, as seen in Figure 2-1. The Drawing Palette and Data Boundary may be hidden or exposed at any time[1]. All other examples showing EUPHORIA in this dissertation keep these parts hidden. In this example, end-user defined widgets are used to represent each display component (e.g., see creation of the thermometer widget in Figure 2-8).

1. In the future, we may also provide a way of locking these parts as hidden, preventing end-users from tampering with completed GUIs.

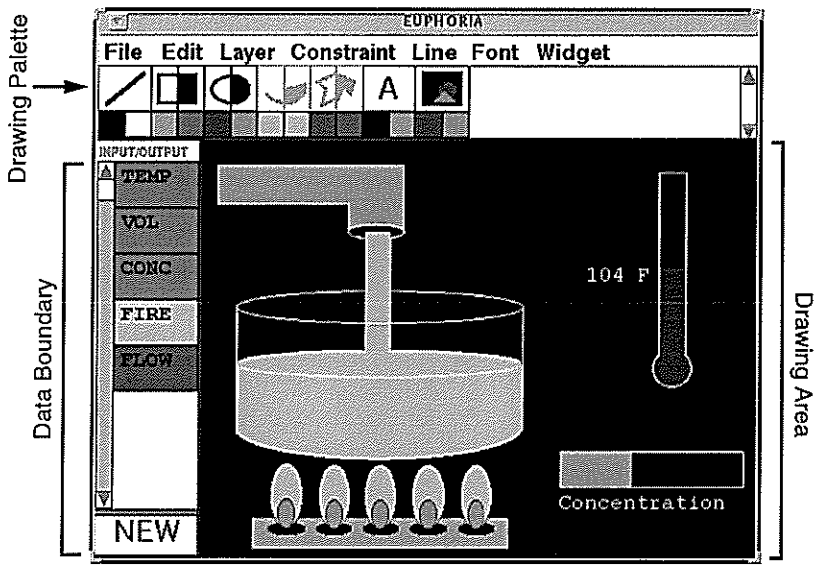


Figure 2-1: Maple syrup factory GUI in EUPHORIA.*

Each widget has an exposed state value which may be manipulated by the end-user and is connected to EUPHORIA's data boundary (see Chapter 3) through the use of constraints. Widgets are drawn using their own coordinate systems, allowing external modules to specify display values in terms of real world values rather than raw pixels. For example, the temperature is specified in terms of degrees Fahrenheit rather than the pixel height of the thermometer's "mercury" rectangle. This allows the application portion to be created completely independent of their user interface.

2.2.2 Remote Control Video Camera

The Remote Control Video Camera application is used to automate the recording of simple video productions. Used in conjunction with a program-controllable video camera, this application periodically updates the camera's angle to track the movement of a subject.

Figure 2-2 shows a GUI for this application created with EUPHORIA[2]. This GUI consists of a wide view, a sample image, and a couple of controls. The wide view is taken in advance representing the extent to which the camera can operate. A rectangle displays the camera's current field of view within this area; the user may manually control the camera by moving and/or resizing the

2. As of this writing, the application modules of the Remote Control Video Camera are still under development. However, EUPHORIA supports all of the functionality described in this section.

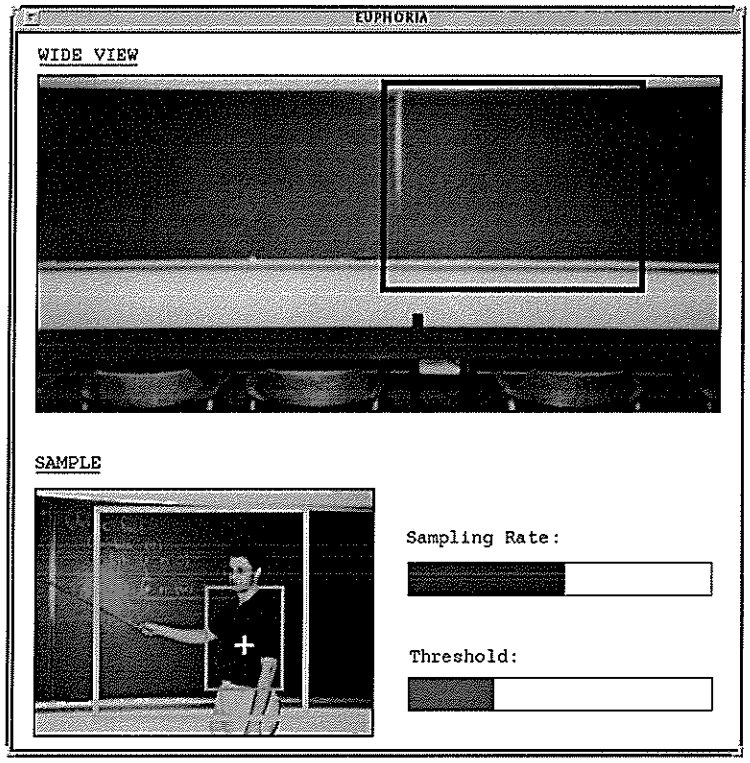


Figure 2-2: Remote Control Video Camera GUI in EUPHORIA.*

rectangle. The sample image represents a recently captured image from the camera, obtained at a rate determined by the "Sampling Rate" slider widget. Superimposed on the sample image are graphics items used to monitor and control the application's shape recognition algorithm (implemented in a separate module). A light blue rectangle represents the extent to which the subject can operate without causing the camera to move. In this example, the image recognition algorithm recognizes the subject based on a target color (i.e., the color of his shirt, blue). A green rectangle is used to show the minimum rectangular bound that the algorithm recognizes as having this color. The user can modify a "Threshold" slider to control the strictness of the color recognition process. Finally, a yellow cross is used to display the center position of the target color's greatest concentration. The user moves this cross initially to inform the recognition algorithm of the target color.

2.3 Related Work

Extensive work has been done in the area of user interface construction. Thinglab [5] uses constraints to specify relationships between parts of a simulation graphical display. Thinglab represents early work in graphical constraint systems and provided the foundation for many later systems. Garnet [48] provides a toolkit which allows the user to construct interactive graphical user interfaces using an object oriented constraint-based library. Garnet does not provide end-users with graphical mechanisms for establishing relationships between the user interface and the application that it controls.

Fabrik [30], [38] and LabVIEW [31] provide self-contained visual programming environments for sequential computation. Fabrik represents visual programs as data flow graphs of connected component icons. "Pins" are used as part of an overall electronics store metaphor, representing data ports used in connecting the components of a visual program. Fabrik's pin and component mechanisms are similar to the "plug and play" nature of Playground's module abstraction (Section 1.5.1), graphics handles, and end-user defined widgets (Section 2.7). LabVIEW is a commercial visual programming environment designed for use by engineers and scientists with little or no traditional programming experience. The basic component of any program in the LabVIEW environment is a "virtual instrument," that consists of a front panel and a block diagram. Programs are written in G, a data flow based language with a special set of additional control flow. Neither Fabrik nor LabVIEW support distributed applications. Both have a fixed set of predefined graphical components. In contrast, EUPHORIA supports end-user construction GUIs for distributed applications out of user-defined graphics components. The computational components of a distributed Playground application are created by programming in an existing language (e.g., C++). Our graphical tools are then used to define relationships among the states of modules in a distributed system and the state of the GUI, as well as relationships among graphics objects within the GUI.

2.4 Graphics Primitives

EUPHORIA supports a number of different types of graphics objects including rectangles, ovals, lines, text, and movies (an image is a movie consisting of a single frame). Figure 2-1 and Figure

2-2 contain examples of each of these graphics object types. Each graphics object has a set of attributes whose values define its visual appearance and other state information. Graphics object attributes are visually represented as *handles* that are positioned in appropriate places relative to the object. Handles serve a dual role in EUPHORIA. In addition to dragging, the user can also establish relationships (i.e., constraints) among the attributes of the graphics objects by using the handles as data ports. The various handles represent different data type information. For example, the "top-left" handle of a rectangle represents an (x, y) tuple of real values; the "content" handle of a text object represents a string value.

2.5 Constraints

Constraints are a simple, yet powerful, way for end-users to specify persistent relationships among graphical object attributes. Once a constraint relationship is formed, the constraint solver is responsible for maintaining the relationship when changes are made to connected graphics objects or published variables.

2.5.1 Basic Constraints

Three types of basic constraints can be established by end-users: anchor, equality, and conversion. *Anchor constraints* are formed on a graphics attribute handle, setting the attribute to be constant. *Equality constraints* are formed by dragging a connection line between two graphics object handles. A *conversion constraint* is a specialized equality constraint in which the graphics attribute data types are compatible, but not the same, requiring a type conversion (e.g., connecting a real number graphics attribute to a string graphics attribute).

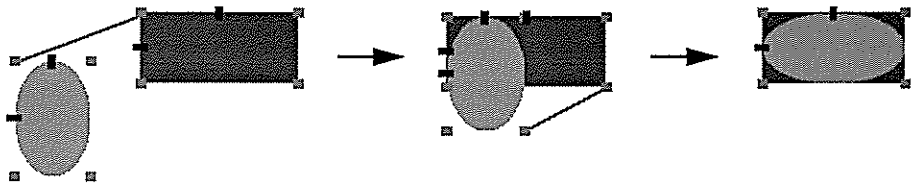


Figure 2-3: Inscribing an oval within a rectangle.

Figure 2-3 shows the process of inscribing an oval within a rectangle through the use of constraints. A constraint between the oval and rectangle left-top corners is formed by dragging a

connection line between the corresponding handles. This action causes the oval to "snap" to the position of the rectangle. A constraint is also formed between the right-bottom corners of the oval and the rectangle, inscribing the oval within the rectangle. These constraint relationships are maintained by the constraint solver; moving or resizing either the oval or the rectangle causes the other to change as well.

2.5.2 Constraint Visualization

EUPHORIA supports visualization and editing of constraints, allowing end-users to optionally view constraints of graphics objects, view current computation directions, delete constraints, and change constraint strengths. Constraint computation directions are helpful in understanding how objects of a constrained drawing interact. An anchor constraint is visually represented as a rectangle surrounding the constraint's corresponding handle. An equality or conversion constraint is visually represented as a directed arrow between graphics object handles (see Figure 2-4). All visualized constraints flash to distinguish them from the surrounding objects. The colors between which the constraints flash indicate the types of values constrained. The thickness of the constraint shows its strength (see Section 1.5.2). If there is not sufficient space to display a directed arrow for an equality or conversion constraint, then the constraint is visually represented as a circle.

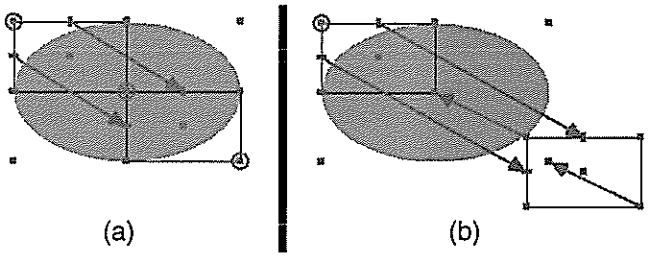


Figure 2-4: Constraint visualization.

Since constraint relationships in EUPHORIA can grow to be complex, the user may want to examine or edit only selected portions of the constraint graph. Constraints among selected graphics objects can be visualized and edited, with constraints to imaginary or off-screen objects hidden unless the user decides to display them. This allows for selective viewing of constraint

information. End-users can see how constraints are enforced, as well as which constraints cannot be enforced (unenforced constraints are represented as undirected, dashed lines).

In certain cases, it may be difficult or impossible to determine to which objects a constraint is connected or the propagation direction of a constraint, especially when a number of points are constrained together (see Figure 2-4a). In these cases, the user may temporarily displace the apparent positions of objects on the screen. However, this editing has no effect on any other objects on-screen; the "real" positions and sizes remain as they were previously. This allows the user to pull graphics objects apart whose corners are constrained together, in order to view, edit, or delete the constraints between them (Figure 2-4b). When the editing is complete, the displacements are reset and the objects "snap back" to their original positions.

2.5.3 Constraint Strength Specification

The strength of a constraint may be specified by an end-user. Constraint strengths are used to establish behavioral preferences among graphical constraint relationships. In the event that constraints conflict or there are cycles, the strengths are used to resolve the conflicts and cycles, favoring stronger over weaker constraints. Manipulation of graphics objects is internally implemented using constraints (see Section 2.8). The strength of various types of manipulation actions may be adjusted, modifying the manipulation behavior.

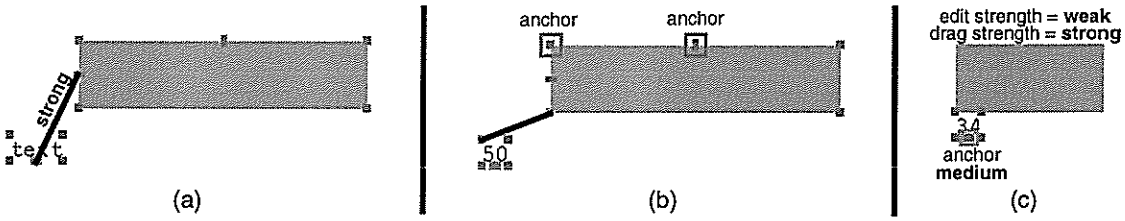


Figure 2-5: End-user creation of an interactive bar graph with a digital read-out.

For example, Figure 2-5 shows the process of creating an interactive bar graph with an associated digital read-out. First, a rectangle and text object are created (Figure 2-5a). A strong conversion constraint is formed between the rectangle's width handle and the text object's string handle. As a result, the text displays the value of the rectangle's width, even during direct manipulation of the rectangle. Second, the text is made adjacent to the rectangle's bottom through an equality

constraint between the positions of the rectangle and the text; the position and height of the rectangle are anchored through the use of constant constraints (Figure 2-5b). At this point, the text may also be edited, changing the width of the rectangle during user typing. However, since the text is only intended to serve as a digital read-out (not an editable label) in this application, the string attribute of the text is then anchored with a medium strength constant constraint (Figure 2-5c). This disables direct editing of the text since the constant constraint's strength (medium) is stronger than the edit text strength manipulation action (weak). However, direct manipulation of the bar graph still results in the textual display of the bar's width since the conversion constraint strength (strong) and the manipulation drag strength (strong) override the anchor's strength (medium).

2.5.4 Imaginary Objects

Many desirable constraint relationships cannot be established by equality constraints directly between graphics object attributes. *Imaginary objects* are invisible shapes that serve as an abstraction for defining indirect constraints between attributes. Any simple or compound graphics object can be made imaginary. The attributes of an imaginary object can be constrained with the attributes of other graphics objects. In this way, users can create indirect constraints among graphics objects visually using the same mechanism used to create constraints among the attributes of visible objects. Note that imaginary objects are not only used in establishing positional relationships; imaginary objects can be used for forming indirect relationships among any attribute type.

For example, suppose that an end-user needed to construct an adjustable picture frame for a photograph. That is, the photograph should be surrounded by a border of uniform thickness and both the photograph and border should be adjustable through direct manipulation. Figure 2-6 illustrates how this frame may be constructed by an end-user through the use of constraints. First, three rectangles representing the frame and border offsets are drawn. The corners of the offsets are attached to the frame by creating constraints among the appropriate graphics object handles (Figure 2-6a). The offsets are instantaneously snapped into place when each constraint is established. Each rectangle may still be independently adjusted, except that the offsets always remain attached to the opposite corners of the frame. Second, the picture is inscribed into the

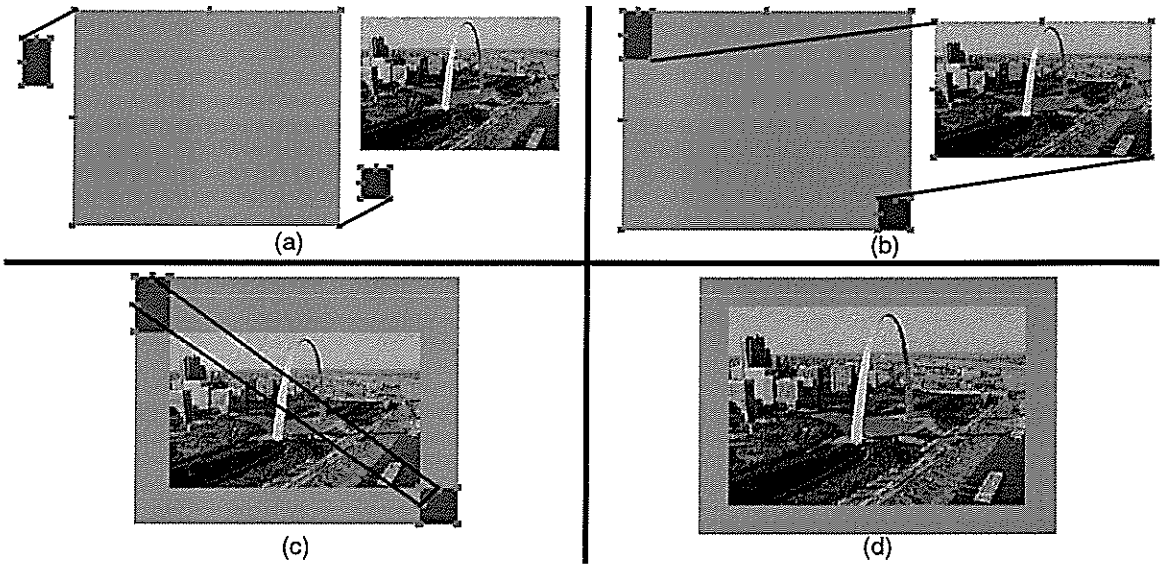


Figure 2-6: End-user creation of an adjustable picture frame using constraints*.

frame by establishing constraints among the inner corners of the offset rectangles (Figure 2-6b). This action snaps the picture into place while still allowing the dimensions of both the picture and the frame to be manipulated. Third, the size of the offset rectangles are constrained to be equal by connecting their width and height handles (Figure 2-6c). Also, the lower offset rectangle is set to be a square by constraining its width and height to be equal. Finally, the offset rectangles are set to be imaginary, resulting in an adjustable frame surrounding the picture (Figure 2-6d). Note that this example involves constraint cycle avoidance since the added constraints cause potential cycles with the rectangles' internal constraints (see Section 2.8).

Although the effect of indirect constraints may also be achieved by simply changing the color of objects to the background color, imaginary objects provide additional expressive power. First, imaginary objects are not actually drawn by the system, which serves to increase efficiency as well as completely hiding the details of indirect constraint relationships. Second, when imaginary objects are not shown, they are not selectable, and therefore do not adversely affect end-user interaction with a completed GUI. Third, the user interface designer has the ability to view or hide imaginary objects, making modifications much easier to perform.

2.5.5 Formula Constraints

A calculator object allows end-users to specify formula constraint relationships among graphics objects. Like other graphics objects, a calculator has a number of attributes, from which the user can make constraints. The attributes are given names, and an editing area allows the user to specify an algebraic formula using the attribute variables. A multi-way constraint graph is constructed from the formula, providing a means to compute any of the variables dynamically in terms of the others. Once the user is satisfied with the configuration, the calculator can be made imaginary (i.e., hidden, see Section 2.6).

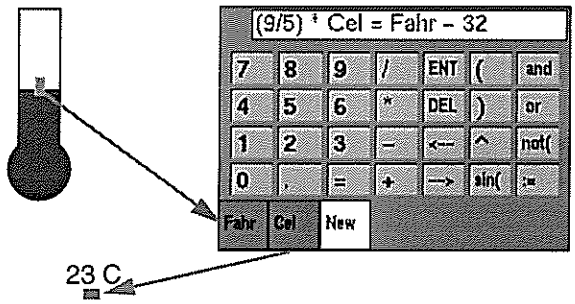


Figure 2-7: A calculator object for converting between temperature scales.

For example, a calculator object could be used to convert between scales of measurement, such as Celsius and Fahrenheit temperatures (see Figure 2-7). The calculator maintains the mathematical relationship between these two variables, computing degrees Celsius when the thermometer is manipulated or computing degrees Fahrenheit when a new Celsius value is entered. This approach enables end-users to specify simple formulas in a more natural way than with conventional visual programming (e.g., ThingLab's temperature converter [5]).

2.6 Spaces

A *space* is a coordinate system that contains graphics objects. For example, the main drawing area of the EUPHORIA window is a space. To simplify user interface construction, EUPHORIA allows end-users to define multiple spaces with independent coordinate scaling factors, origins, and bounding rectangles. User defined coordinate systems allow a user interface to be defined in terms of meaningful values instead of raw pixels. In this way, the external application portion does not

need to be aware of how it GUI displays information, greatly simplifying the application's implementation and increasing its reusability.

A space can have multiple representations called *alternatives*. For example, a simulation GUI might consist of an alternative that shows the simulation state graphically, allowing direct manipulation, and an alternative that shows expanded information in a more textual representation. Alternatives are most commonly used to construct widgets that have a changing graphical representation.

2.7 Widgets

A widget is an encapsulated space containing graphics objects with a *data boundary* of published attributes. Widgets are created graphically by end-users. The data boundary defines the subset of attributes that can be used externally to control its appearance. As with other graphics objects, the external attributes of a widget can be viewed, revealing handles that can be used for direct manipulation or forming constraint relationships.

A special tool is not required to create widgets. Instead, a widget can simply be drawn in EUPHORIA and saved. One can load a EUPHORIA file, interpreting it as a widget. In this way, users do not need to learn about a different interface for creating widgets and the implementation does not need to support two different types of graphics editors.

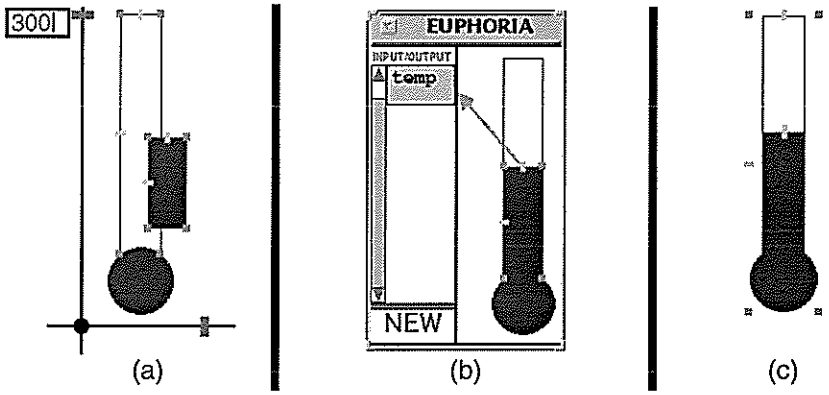


Figure 2-8: Creating a thermometer widget.

For example, the thermometer widget from Section 2.2.1 can be constructed by an end-user as follows. First, the component shapes of the thermometer are drawn and constraints among the shapes are formed (Figure 2-8a). A scaling factor for the widget space is set using a coordinate system tool so that the top of the thermometer represents 300 degrees Fahrenheit. This allows external applications to interact with the thermometer in terms of real world values rather raw pixels. Second, the data boundary of the widget is defined by publishing the height of the mercury (i.e., the temperature, see Figure 2-8b). The data boundary specifies that only the temperature attribute will be exposed from the widget when it is used. The specification of the widget is saved, and the widget can then be used within a GUI (Figure 2-8c). When the widget is used, only the handles representing the published attributes and the default bounding box are available.

Alternatives

A widget can also have alternatives that show different representations. The end-user creates and labels these representations within EUPHORIA using conventional drawing techniques. The choice of which alternative is displayed is controlled through the use of an alternative ID handle, that can be connected to other graphics object attributes or EUPHORIA's data boundary.

Figure 2-9: Alternatives of a vertex widget, representing different states.

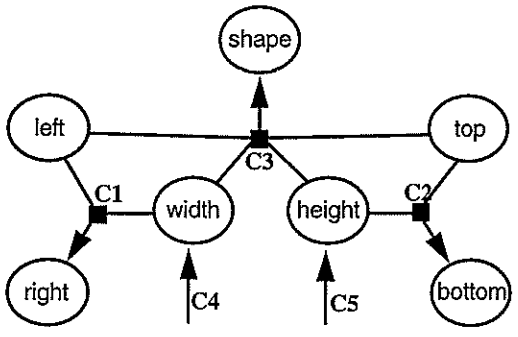
For example, Section 3.2.2 describes a visualization of a distributed minimum spanning tree algorithm. Each vertex of the graph is represented as a widget with a textual display for the "level number" attribute used by the algorithm. In the algorithm, each vertex also has a "state" attribute which can take on the values of *sleeping*, *find*, or *found*. The widget visually represents the state attribute using alternatives, as seen in Figure 2-9. In the sleeping state, the vertex is represented as a blue circle without the level attribute displayed. In the find state, the vertex is represented as a red square with the level number shown textually. In the found state, the vertex is represented as a green circle with the level number shown textually.

2.8 Constraint Architecture

In addition to maintaining end-user defined relationships, constraints are an important part of EUPHORIA's internal implementation. Constraints are used to represent the internal relationships among window components (i.e., window layout), graphics objects, and graphics object handles. UltraBlue's value consistency and cycle avoidance features (see Chapter 4) are needed to support this functionality. Through this uniform constraint mechanism, the implementation of EUPHORIA is greatly simplified. This section describes some of these constraint applications in detail. See Section 1.5.2 for a description of the constraint visual notation used within this section.

2.8.1 Graphics Object Representation

Internally, the relationships among the graphics object attributes are represented as a constraint graph. Figure 2-10 shows the constraint graph for a rectangular bounded shape (e.g., the rectangles and images from Figure 2-2), with variables for attributes such as left, right, width, etc. Constraints and their associated strengths are used to maintain relationships and the behavior of the shape. Constraints C1 and C2 compute the sum function among the shape's dimension variables. These multi-way constraints allow any one of their associated variables to be computed in terms of the others. The strength of these constraints is "required" since they should always be maintained in the presence of conflicting constraints. In addition, the width and height variables have value consistency assertions (see Section 4.4.5) that prevent their values from becoming negative.



	strength	*function*
C1	required	left + width = right
C2	required	top + height = bottom
C3	required	Draw with dimensions: (left, top, width, height)
C4	weak	No operation.
C5	weak	No operation.

Figure 2-10: Constraint graph for a rectangular bounded shape.

Constraint C3 is known as an *active value* constraint [25]. The purpose of an active value constraint is to perform a "side effect" action whenever it is executed. In this case, C3 is used to draw the rectangular shape whenever one of its dimension variables is modified[3]. Since C3 represents a side-effect action, and not an equation, it is represented as a one-way constraint (see Section 4.2.1) with "shape" as its only output variable. This variable contains an implementation dependent representation (i.e., a pointer to a rectangle class with associated window system information) that is used in drawing the shape.

Constraints C4 and C5 are known as *stay* constraints. The purpose of a stay constraint is to state the preference that a variable should remain constant in the absence of stronger, conflicting constraints. Stay constraints are used to specify the behavior of a constraint graph when constraint relationships are underconstrained, reducing nondeterministic behavior. As with other constraints, the strength of a stay constraint represents its importance. In this case, C4 and C5 are of strength "weak," keeping the width and height variables constant in the absence of stronger conflicting constraints. The stay constraints on width and height keep the size of a shape constant as it is moved.

2.8.2 Manipulation

As mentioned earlier, shapes are manipulated through the use of constraints. Whenever a rectangular shape is moved, *edit* constraints are added to some of its variables. A variable with an enforced edit constraint is not computed in terms of any other constraint or variable, allowing the system to safely set its value externally. In this case, as the shape is dragged by the user, its associated position is placed into the left and top variables. These values are propagated through the constraint graph during execution, resulting in constraint computations and active value side-effects. That is, the values of related quantities such as the "right" and "bottom" attributes are computed, and the connected shape(s) are redrawn. In this way, complex series of connected shapes may be manipulated in real-time.

---

3. The shape is invalidated for later redrawing [43].

2.8.3 Graphics Object Handles Representation

The constraint graph of a graphics object can be connected directly to the constraint graphs of its associated handles. In this way, one can manipulate a graphics object using the same mechanism that is used for other types of internal communication, simplifying the EUPHORIA implementation. Communication among graphics components and the external environment occurs automatically as a by-product of manipulating a graphics object. Figure 2-11a shows the constraint graph representation of a rectangle with an attached handle on its left-top corner. Attaching a handle simply involves creating a constraint graph for the handle and forming constraints between the handle's graph and the rectangle's graph (see Figure 2-11a, constraints C1 and C2). In this way, whenever the rectangle or the handle change position or size in any way (e.g., direct manipulation, external interprocess communication, etc.) the other is changed and redrawn to be consistent with that change.

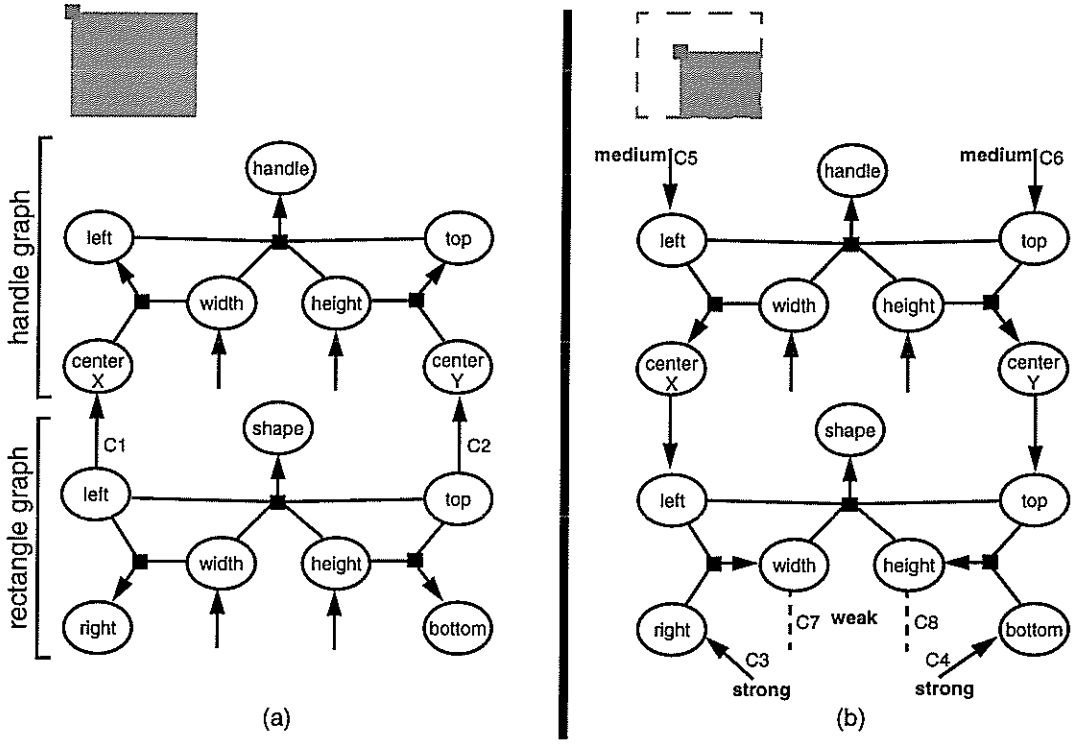


Figure 2-11: Graphics object handle constraint graph.

To resize the rectangle, a few other constraints are added to the graph. In Figure 2-11b, the right and bottom constraint variables are anchored with strong constant constraints (C3 and C4). The

left and top handle constraint variables are set editable by attaching medium edit constraints (C5 and C6). The strengths of these constraints are stronger than that of the stay constraints on the size of the rectangle (C7 and C8, weak strength). The stay constraints are unenforced in favor of the edit constraints, resulting in a redirected constraint graph that computes the rectangle's size in terms of the handle's position. Whenever the handle is moved by the user, the x and y coordinates of the handle are copied into the handle's left and top constraint variables. Execution of the constraint graph results in: (1) redrawing the handle, (2) computing a new size for the rectangle, (3) redrawing the rectangle, and (4) propagation of values to any other objects that may be attached to the graph.

Other handles are added to the rectangle's constraint graph in a similar way. No special purpose programming was required to create each individual handle. Other handles use an identical constraint graph structure, but are connected to different variables of the rectangle's constraint graph (e.g., a right-bottom handle is created by attaching the handle's constraint graph to the right and bottom variables rather than the left and top variables shown in Figure 2-11).

This constraint graph approach was later augmented to support a number of other features such as constraint visualization and type conversion. The displacement mode described in Section 2.5.2 was implemented by adding additional "offset" variables to the rectangle's constraint graph, in order to distinguish between the real position of a rectangle and its apparent position. Active value constraints are used to visualize end-user defined constraints.

2.9 Summary

EUPHORIA is a user interface management system that we have developed to explore the following user interface construction techniques. First, end-users can define constraints, including direct constraint relationships (anchor, equality, conversion) and indirect constraint relationships (imaginary objects, formula constraints). Second, end-users can visualize and edit these constraint relationships. Third, end-users can define their own coordinate systems, enabling the external application portion to be created independent of its graphical representation(s). Fourth, end-users can define widgets with independent coordinate systems and alternative representations. These capabilities, along with the interprocess data visualization and manipulation described in Chapter

3, enable end-users to construct complete direct manipulation graphical user interfaces for distributed multimedia applications. An end-user usability study was conducted (Chapter 6), showing that end-users are able to comprehend and apply these techniques.

Chapter 3

Interprocess Data Visualization and Manipulation

Chapter 2 discussed how end-users can construct the components of a direct manipulation GUI and establish relationships among the various graphical components. The next step in constructing a complete GUI is to associate the user interface components with their underlying application, allowing the application to drive the behavior of the GUI and to respond to user interaction. Enabling end-users to specify the interaction between a GUI and its underlying application has been a long standing goal of user interface research. It is very difficult to give this power to the end-user. For non-trivial distributed applications, the application portion consists almost entirely of textual source code that the end-user is unlikely to understand. Indeed, most user interface development systems only allow the programmer to specify the interaction between the application and its GUIs, typically by specifying function calls directly to the application code.

As described in Section 1.3.1, The Programmers' Playground is used to construct the application portion of a distributed application. With Playground, programmers create general-purpose modules through the use of a software library (Section 1.5.1). End-users construct distributed multimedia applications by creating GUIs with EUPHORIA and configuring communication among the modules and the GUIs through the use of a visual configuration language that we have developed [42]. This chapter describes techniques that enable end-users to define a GUI in terms of an I/O Abstraction module, facilitating communication among the application portion and its GUI(s). In addition, aggregate visualization and manipulation techniques have been developed to enable the construction of GUIs with dynamically changing collections of graphical components. *Aggregate mappings* enable end-users to define interactive visualizations of aggregate elements

(see Section 3.5), providing mechanisms similar to a relational database's project, select, join, and cross product operations [9], [10].

3.1 Contributions

1. End-user techniques for exposing a GUI's state components to the external environment of a distributed application through a variety of data types (real, string, tuple, array, etc.).

2. An end-user mechanism for defining and instantiating new tuple and aggregate data types at run-time.

3. An end-user mechanism for the visualization and manipulation aggregate data, including mechanisms similar to a relational database's project, select, join, and cross product operations.

4. An architecture for performing fast join and cross product operations, allowing efficient incremental updates.

3.2 Motivating Examples

This section presents two example applications that utilize some of the key user interface construction techniques described within this chapter.

3.2.1 Image Morphing

Image morphing is the process of transforming one image into another, forming a series of intermediate images that animates the metamorphosis. Morphing is commonly used in the entertainment industry to produce special effects for movies and television. This application allows the user to establish a morphing operation between two images by specifying corresponding features on each of the images.

For example, Figure 3-1 shows the GUI and partial output of the image morphing application constructed in EUPHORIA. In this example, a morph is established among the Star Wars characters Chewbacca and Yoda [69]. Two aggregate mappings (Section 3.5) are used to create numbered point widgets on the start and finish images (i.e., the first aggregate mapping creates the set of points for the start image, a second aggregate mapping creates the set of points for the finish